Análise Comparativa do Desempenho de Comunicação em Nós Remotos Usando MPICH2 ou MPIJava

Murilo da S. Dantas[1,3], Ariovaldo de S. Junior[3], Daniel A. Cesário[3], Ricardo K. de O. Barros[3], Luis G. U. Ungur[3], Nilson Sant'Anna[2], Reinaldo R. Rosa[2]

[1]Programa de Doutorado em Computação Aplicada – CAP
Instituto Nacional de Pesquisas Espaciais – INPE

[2]Laboratório Associado de Computação e Matemática Aplicada – LAC
Instituto Nacional de Pesquisas Espaciais – INPE

[3]Tecnologia em Informática
Faculdade de Tecnologia de São José dos Campos – FATEC

{murilodantas06,ariovaldojunior,danielatcesario,rkramer1988}@gmail.br,
lgungur@hotmail.com,{nilson,reinaldo}@lac.inpe.br

Abstract. *In the development of robust scientific applications involving parallel computing, if the choice of programming environment is not careful, applications can be influenced by the low performance offered through the environment. This happens because there are several techniques of parallelism and each language implements its using industry standards, or not. An important criterion for the choice of programming environment for parallelized applications is the evaluation of their libraries. The assertiveness of the environment choice certainly influences the improved performance of the application as a whole.*

Resumo. *No desenvolvimento de aplicações científicas robustas que envolvam computação paralela, se a escolha do ambiente de programação não for criteriosa, as aplicações podem ser influenciadas pelo baixo desempenho oferecido através do ambiente. Isso acontece, pois existem várias técnicas de paralelismo e cada linguagem implementa a sua usando padrões da indústria, ou não. Um critério relevante para a escolha do ambiente de programação para aplicações paralelizadas é a avaliação de suas bibliotecas. A assertividade da escolha do ambiente certamente influencia na melhora do desempenho da aplicação como um todo.*

Palavras-chave: desempenho, comunicação, MPICH2, MPIJava, ambiente virtual.

1. Introdução

É comum o uso de técnicas de paralelização de código na solução computacional de um problema científico que exija desempenho no seu processamento. Atualmente, existem muitas linguagens computacionais que proveem paralelização através de bibliotecas específicas para esse fim. A escolha da linguagem computacional para o desenvolvimento de aplicações paralelizadas pode influenciar significativamente no desempenho das mesmas.

Uma linguagem computacional pode oferecer vários recursos que influenciem no desenvolvimento da aplicação. A linguagem Java fornece muitos benefícios como o

bom gerenciamento de memória, a portabilidade sem necessidade de recompilação de código-fonte, a arquitetura multiplataforma e a orientação a objetos. Para que alguns desses benefícios sejam possíveis, Java possui em sua arquitetura uma camada adicional de software, a *Java Virtual Machine* (JVM). Essa abstração pode influenciar sensivelmente no desempenho da aplicação, conforme [Yalamanchilli e Cohen 1998], levando-a a apresentar uma performance um pouco inferior, quando comparada às linguagens de mais baixo nível como C e Fortran, já que estas permitem que a aplicação seja compilada diretamente na linguagem da máquina. Escolher uma linguagem para o desenvolvimento de uma aplicação mais robusta se torna, portanto, uma decisão estratégica no sentido de equilibrar as demandas de produtividade *versus* desempenho.

Algumas soluções de paralelismo usam o *Message Passing Interface* (MPI) para a comunicação entre os processadores ou nós de um *cluster*. O MPI é um padrão bem difundido e permite que diversos processos de uma aplicação interajam de maneira eficiente usando troca de mensagens. Muitas linguagens possuem bibliotecas que fornecem rotinas definidas em MPI e vários ambientes computacionais para computação científica fazem uso do MPI em rotinas que permitem paralelização.

Neste artigo, são avaliadas duas bibliotecas para paralelização usando esse padrão: a MPICH2 para a linguagem computacional C e a MPIJava, para a linguagem Java. No experimento é avaliada a comunicação entre processos remotos usando algoritmos do tipo *ping-pong*. Em tais algoritmos, um processo envia uma quantidade de dados para outro através da biblioteca. Para medir o desempenho, o critério escolhido foi obter o tempo de *round-trip*, que é o tempo para uma mensagem ir até um nó escravo e voltar. Com isso, o objetivo da pesquisa foi de mensurar as diferenças de performance entre bibliotecas de paralelização de cada linguagem, determinando qual solução possui o melhor tempo de resposta e se esta diferença de desempenho é muito significativa.

Uma breve introdução sobre aplicações paralelizadas usando MPI é dada na Seção 2. Na Seção 3 são descritos o experimento e os programas utilizados. Já na Seção 4 são apresentados os resultados dos testes e, finalmente, na Seção 5 é feita uma discussão acerca de tais resultados.

2. Aplicações Paralelizadas

A computação paralela ou paralelismo é uma prática que possibilita obter alto desempenho através da combinação de máquinas individuais ou múltiplos processadores em uma mesma máquina. Uma tarefa complexa é dividida em várias tarefas menores que são processadas em paralelo em várias máquinas, o que torna possível a conclusão da tarefa em menos tempo.

Com a redução no preço dos microcomputadores, ao mesmo tempo em que estes apresentam capacidade de processamento cada vez maior, está cada vez mais acessível construir computadores poderosos a partir de máquinas de baixo desempenho. Podem ser citados alguns exemplos, como o supercomputador GridUNESP [GridUnesp 2011] que conta com 2944 núcleos de processamento; e o *cluster* que foi utilizado no filme Titanic, que utilizou 200 PCs comuns DEC Alpha em conjunto com algumas estações da Silicon Graphics para a renderização dos efeitos especiais [Bertozzi et al 1998], sendo um dos pioneiros na utilização deste recurso na indústria cinematográfica.

Vários tipos de aplicações se beneficiam deste recurso. Universidades utilizam *clusters* de computadores para obter cálculos científicos nos diversos campos do conhecimento, como por exemplo, modelagem molecular com o software Gaussian [Keller 2008]. Empresas também se utilizam do recurso da paralelização no desenvolvimento de jogos [Kerlow 2004]. Entre as áreas que se beneficiam da

computação paralelizada podemos citar modelagem climática para a previsão do tempo, ressonância magnética, dinâmica de fluidos [Kennedy e Koelbel 2003], gerenciamento de consultas em sistemas de banco de dados [Tania et al 2008], simulações diversas, entre outras.

Em aplicações paralelizadas desenvolvidas em linguagens tradicionais como C ou Fortran é comum a necessidade da recompilação do código-fonte no caso do uso de máquinas com diferentes sistemas operacionais ou arquiteturas. Assim, em soluções de MPI para paralelização em linguagem C é necessário que a estrutura física (os computadores) seja homogênea [Yalamanchilli e Cohen 1998], caso contrário será necessário gerar um executável para cada máquina diferente empregada. Num ambiente heterogêneo, a solução pode ser o uso de uma linguagem que abstrai a infraestrutura a partir do sistema operacional através do conceito de virtualização. A linguagem de programação Java é uma candidata para sanar esse problema.

A MPI é uma biblioteca que permite a paralelização de uma aplicação usando troca de mensagens [Gropp et al 1998]. Suas versões originais foram implementadas em linguagem C e Fortran e esta abordagem permite que a comunicação ocorra de forma muito eficiente, podendo ser implantada em ambientes heterogêneos mantendo a sua confiabilidade [Aoyama e Nakano 1999] e [Hafeez et al 2011]. Além disso, o sistema abstrai do usuário os problemas relacionados a falhas de comunicação.

3. Avaliação das Bibliotecas

Para avaliar a comunicação entre processos usando MPI nas linguagens C e Java, foram escolhidas as bibliotecas MPICH2, para C, e MPIJava para Java. Essas bibliotecas foram escolhidas seguindo os critérios de serem compatíveis com a especificação padrão MPI e serem bastante difundidas no meio acadêmico.

A MPICH2 foi desenvolvida em parceria por cientistas patrocinados pelo Argonne National Laboratory (Divisão de Matemática e Ciências da Computação) e pela Mississipi State University (Departamento de Ciências da Computação). O CH do nome desta solução se refere à palavra *Chamaleon* (camaleão em inglês), e foi adotado por seus desenvolvedores como forma de destacar sua adaptabilidade e portabilidade [Gropp et al 1998]. Já a solução Java escolhida, a MPIJava, foi desenvolvida pelo Pervasive Technology Labs, que é um grupo de pesquisa da Universidade de Indiana. É uma interface orientada a objetos e utiliza código nativo em C em parte de sua implementação [Baker et al 1998].

Como experimento de *benchmark* foram empregados dois algoritmos de *ping-pong* fornecidos para MPI: um para C pela Trac Integrated SCM & Project Management [Trac 2011] e outro para Java pela Pervasive Technology Labs, mesmo laboratório que desenvolveu o MPIJava [Carpenter 2007]. Estas ferramentas foram utilizadas para medir o desempenho da comunicação básica nos testes e ambas foram ligeiramente modificadas de forma a se conseguir a melhor correspondência possível entre os dois códigos. O funcionamento destas soluções se baseia no envio de mensagens de variados comprimentos entre dois processos.

A quantidade de tempo requerido para enviar a mensagem de um processo ao outro deve ser igual ao tempo requerido para enviar esta mensagem de volta. Então, assume-se que o tempo de envio é a metade do tempo de *round-trip*, que é o tempo requerido para a mensagem ir de um processo ao outro e voltar. Veja a Figura 1 para um esquema da disposição do experimento.

Assim, o teste de *benchmark* foi constituído de uma aplicação com dois processos, um principal e outro secundário. Cada processo foi executado em um computador diferente. Quando o processo principal começa a executar na máquina local

ele ativa o processo secundário na máquina remota. O processo principal então cria pacotes sequenciais de mensagens com um número variável de bytes. A primeira mensagem transferida é de um byte e a cada mensagem sucessiva a quantidade de dados é multiplicada por dois até atingir 1048576 bytes, totalizando 21 mensagens. A operação foi repetida 2000 vezes para se obter uma média do tempo de comunicação de acordo com a quantidade de bytes transferida.

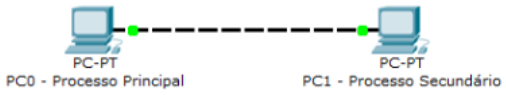

Figura 1. Disposição dos equipamentos utilizados no experimento

Os testes foram executados em dois computadores: o primeiro com processador *Core 2 Duo* de 2.0 GHz e arquitetura de 32 bits com 3 GB de memória RAM; o segundo possui um processador *Dual Core* de 2.0 GHz e arquitetura de 32 bits com 4 GB de memória RAM. Estes computadores estavam conectados por suas interfaces de rede de 10/100 Mbps e por um cabo de par trançado categoria 5e *crossover*. Ambas as máquinas utilizaram o Ubuntu 11.04 como sistema operacional. Todos os códigos escritos em Java foram executados pela distribuição Java Development Kit 7. A versão da solução MPICH2 utilizada foi a 1.3.1 e a da MPIJava foi a 1.2.5.

4. Resultados

Como pode ser visualizado na Figura 2[1], a média do tempo de *round-trip* usando a biblioteca MPIJava é cerca da metade, em quase todos os casos, do MPICH2. Isso contradiz a expectativa de que a solução na linguagem C tivesse maior desempenho. De acordo com os resultados, levando em consideração todas as transferências de tamanhos variados, a MPIJava executa 100 milissegundos mais rápido do que o MPICH2 na média. Entretanto, a partir da transferência de 16384 bytes nota-se uma convergência entre os tempos das duas bibliotecas. Com 1048576 bytes os tempos de comunicação são praticamente iguais.

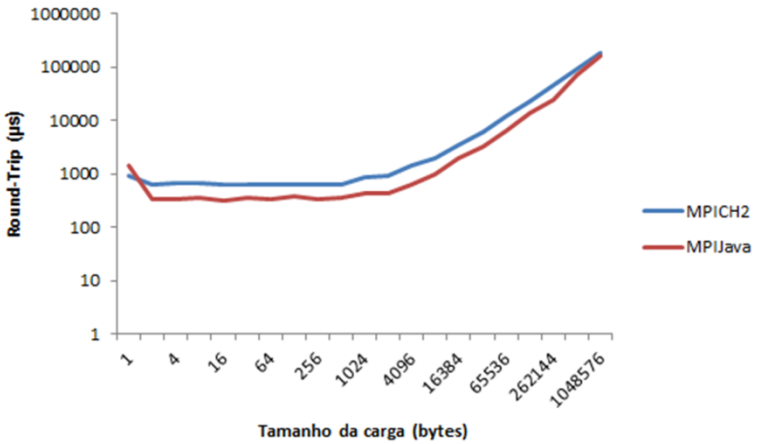

Figura 2. Tempo médio de *round-trip* entre as diferentes bibliotecas

Analisando individualmente as transferências, foi observado que a única carga em que o

[1] Na Figura 2, o eixo vertical do gráfico está em escala log10.

desempenho da implementação em linguagem Java obteve desempenho menor que em C foi a primeira de cada execução dos testes. Isto se deve ao fato de que a JVM faz a tradução do *bytecode* nesse instante. Uma vez que o código é traduzido ele fica na memória até que os testes terminem, e isto contribui para a diminuição dos tempos de *round-trip* seguintes.

5. Conclusões

Num ambiente virtual para computação científica, é comum o uso de infraestrutura que permita paralelização de tarefas. Muitas linguagens computacionais oferecem diversas técnicas para paralelismo nas aplicações que permitam o incremento do desempenho das mesmas. Porém, cada linguagem possui suas peculiaridades que influenciam ora na produtividade de construção do software, ora na segurança do sistema, ora no desempenho da aplicação, seja na manipulação dos dados, ou no processamento de tarefas. Avaliar as estruturas que tais linguagens oferecem para uso pode influenciar nos requisitos de uma aplicação.

As linguagens mais tradicionais como C e Fortran são destacadas por supostamente fornecer maior performance às aplicações nelas desenvolvidas, já que são de mais baixo nível. Já linguagens como Java se sobressaem no quesito produtividade e portabilidade da aplicação.

Neste trabalho, foi avaliada a diferença de desempenho na comunicação entre duas bibliotecas, a MPICH2 da linguagem C e a MPIJava, da Java. Contrariando às expectativas devido à sua estrutura usando máquina virtual, a MPIJava foi melhor na média do que a MPICH2. Esse resultado levanta questões importantes a serem investigadas.

Assim, estudos complementares são necessários para se ter um panorama mais aprofundado das diferenças de desempenho entre linguagens computacionais, como: a avaliação de como tais linguagens gerenciam a memória para os processos e a interação entre eles através das bibliotecas envolvidas; a análise de mais bibliotecas de paralelização; e a análise de bibliotecas de outras linguagens.

Referências

Aoyama, Y., e Nakano, J. (1999) “RS/6000 SP: Practical MPI Programming”, IBM Corporation Red Books, United States of America.

Bertozzi, M., Chiola, G., Ciaccio, C., Conte, G., Marenzoni, P., Poggi, A., e Rossi, P. (1998) “Report on the state-of-the-art of PC Cluster Computing”, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.8666

Carpenter, B. (2007) “Ping-pong test for MPIJava”. Disponível em: http://www.hpjava.org/register.html. Acessado em: 11/2011.

GridUnesp (2011). “Infra-estrutura do GridUnesp”. Disponível em: http://unesp.br/gridunesp/conteudo.php?conteudo=1037. Acessado em: 11/2011.

Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W. e Snir, M. (1998) “MPI, the complete reference”, MIT Press, United States of America.

Hafeez, M., Asghar, S., Ahmad, U. M., Rehman, A. e Riaz, N. (2011) “Survey of MPI implementation” In: Digital Information and Communication Technology and Its Applications: International Conference, Edited by Cherifi, H., Zain, M. E. e Qawasmeh, E., Springer, Germany.

Keller, J. (2008) "Introduction to Gaussian: Computational chemistry using the Arctic Region Supercomputing Center installation of Gaussian 03" University of Alaska Fairbanks. United States of America.

Kennedy, K. e Koelbel, C. (2003) "Languages and Compilers" In: Sourcebook of Parallel Computing, Edited by Jack Dongarra, Morgan Kaufman Publishers, United States of America.

Kerlow, I., V. (2004) "The art of 3D computer animation and effects" John Wiley & Sons, Inc, 3rd Edition. United States of America.

Tania, D., Leung, C. H. C., Rahayu, Wenny. e Goel, S. (2008) "High-performance parallel database processing and grid databases", John Wiley & Sons, Inc., United States of America.

Trac Integrated SCM (2011) "Ping-pong test for MPI". Disponível em: http://trac.mcs.anl.gov/projects/mpich2/browser/mpich2/branches/release/mpich2-1.4/src/mpix/armci/tests/mpi. Acessado em: 11/2011.

Yalamanchilli, N. e Cohen, W. (1998) "Communication Performance of Java-based Parallel Virtual Machines" In: Concurrency: Practice and Experience, John Wiley & Sons, Ltda., United States of America.